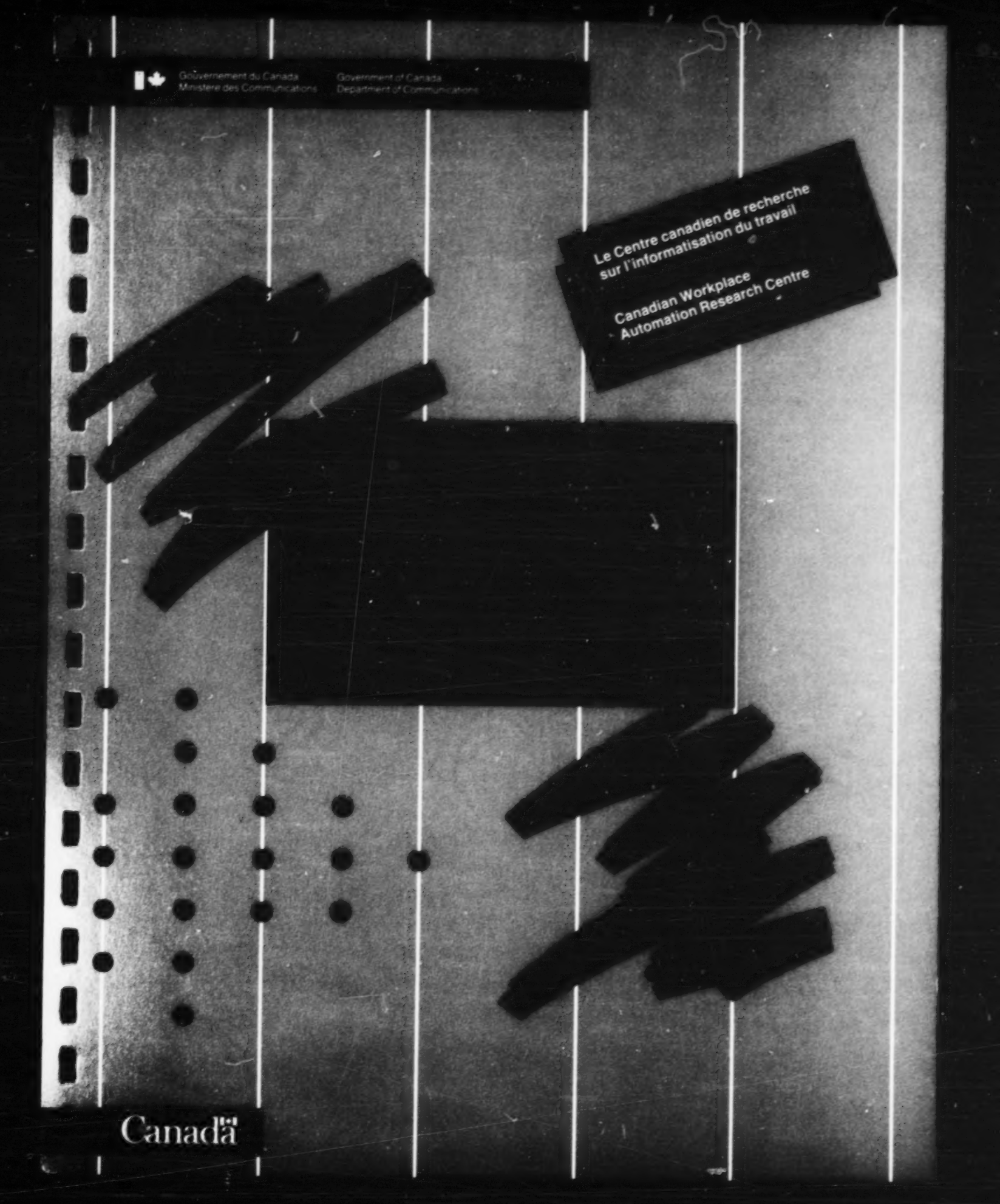
Gouvernement du Canada
Ministère des Communications
Government of Canada
Department of Communications
Le Centre canadien de recherche sur l'informatisation du travail
Canadian Workplace Automation Research Centre
Canada

GRAMMAIRES LOGIQUES RÉVERSIBLES POUR LA TRADUCTION AUTOMATIQUE

Marc Dymetman
Pierre Isabelle

Centre Canadien de Recherche sur
l'Informatisation du Travail

Ministère des Communications Canada

Laval

Novembre 1988

N° de cat: Co 28-1/28-1989 F

ISBN: 0-662-95424-6

Ce rapport a été présenté sous forme de communication à la Second International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages, June 12-14, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Les opinions exprimées dans ce rapport n'engagent que ses auteurs.

* This report is also available in English.

90-04194/

Résumé

Le système de traduction CRITTER se sert d'une seule grammaire pour exécuter les tâches d'analyse et de synthèse. Le formalisme utilisé est une variante des DCGs (Definite Clause Grammars ou Grammaires de Clauses Définies) auxquelles des annotations ont été ajoutées pour permettre une double compilation de la grammaire en deux programmes Prolog d'analyse et de synthèse partageant le même contenu déclaratif.

Ces annotations sont de deux types : 1) annotations séparant le contenu déclaratif des règles (*logique*) de l'ordre de traitement des buts (*contrôle*) et 2) annotations qui servent de directives permettant aux compilateurs d'exécuter des transformations d'"optimisation" sur des groupes de règles, rendant ainsi mieux adaptées à la tâche en cours (*analyse* ou *synthèse*) les procédures Prolog cibles.

Domaines d'intérêt: Grammaires Logiques, Traduction Automatique.

1. LE MODELE DE TRADUCTION

1.1 CRITTER

CRITTER est le système expérimental que nous sommes en train de développer comme banc d'essai pour notre modèle de la traduction. Il est conçu pour traduire de l'anglais au français (et inversement) des rapports sur le marché de la viande produits de façon hebdomadaire par le ministère canadien de l'Agriculture.

Les phrases suivantes donnent un aperçu de la langue de ces rapports :

Imports of slaughter cattle from the United States last week dropped 62% compared to the previous week, totalling 334 steers and 50 heifers.

La semaine dernière, les importations de bovins d'abattage ont chuté de 62% en regard de la semaine précédente, totalisant 334 bouvillons et 50 taures.

1.2 Traits principaux du modèle de traduction.

Nous allons récapituler brièvement les traits principaux du modèle de traduction. Pour une description plus détaillée, voir (Isabelle et al., 1988).

Le modèle de traduction peut être vu comme donné par la composition de trois relations :

- la relation analyse/synthèse source : *anasynt_s(T S, SurfSyn_S, Sem_S)* qui définit un ensemble de triplets bien formés où T_S est un texte en langue source, SurfSyn_S et Sem_S sont respectivement une structure syntaxique de surface et une structure sémantique pour ce texte, toutes deux dépendantes de la langue source;

- la relation analyse/synthèse cible : *anasynt_t(T T, SurfSyn_T, Sem_T)* qui est l'analogue de anasynt_s pour la langue cible.

- la relation de transfert : *tr(Sem_S, Sem_T)* qui définit un ensemble de couples où Sem_S et Sem_T sont respectivement des structures sémantiques source et cible qui sont considérées comme équivalentes du point de vue traductionnel.

Sur le plan formel, les relations *anasynt* sont décrites dans le cadre des *grammaires à clauses définies* (DCG) (Pereira, Warren 1980), auxquelles des *annotations de contrôle* ont été ajoutées (voir la section 2.2).

Quant à la relation tr, elle est définie à l'aide d'un ensemble de clauses obtenues par la compilation en Prolog d'un *dictionnaire de transfert*.

1.3 Structures sémantiques.

Prenons comme exemple la phrase anglaise (assez artificielle):

In Edmonton, John gave hogs to Mary.

En ne tenant pas compte pour l'instant de la structure syntaxique de surface, anasynt_s attribuera la structure sémantique suivante à cette phrase :

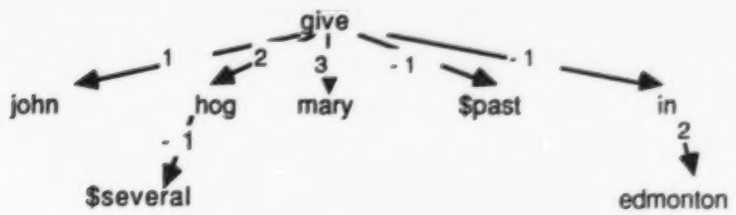

Fig.1

Les étiquettes 1, 2 et 3 peuvent être lues comme des positions d'arguments relatives au prédicat 'give'. Pour ce qui est des étiquettes (-1) ("arguments inverses"), il s'agit d'un dispositif notationnel qui nous permet de représenter simultanément des relations *prédicat-argument* ainsi que des relations de *subordination*.

Par exemple, le noeud 'in' a deux arguments : le premier (-1) étant l'événement correspondant au don, le second (2) étant le lieu de cet événement. La notation "inverse" du premier argument est une façon de rendre la structure sémantique réminiscente de ce que "in Edmonton" est syntaxiquement un dépendant de 'give'. (Pour une justification de cette approche, voir (Isabelle et al., 1988)).

2. Grammaires Logiques Réversibles.

2.1 Buts.

Considérons de nouveau la relation *anasynt* (pour plus de spécificité, nous allons traiter la relation analyse/synthèse source, mais tout cela s'appliquera

mutatis mutandis à la relation cible). La lecture *déclarative* de la relation anasynt_s est simplement qu'il s'agit d'un ensemble de triplets de la forme : (T,SurfSyn,Sem). Dans une perspective *computationnelle*, le défi posé par le problème de la réversibilité est celui d'écrire un *programme* P ayant les propriétés suivantes :

- la donnée en entrée d'un certain texte T aura pour effet la production en sortie de chaque Sem tel que:

  ∃SurfSyn anasynt(T,SurfSyn,Sem)

-.la donnée en entrée d'un certain Sem aura pour effet la production en sortie de chaque texte T tel que:

  ∃SurfSyn anasynt(T,SurfSyn,Sem)

Par exemple, la donnée en entrée au programme de "*In Edmonton, John gave hogs to Mary*" devrait avoir pour résultat la sortie de la structure sémantique Sem de la figure 1 et la donnée en entrée de Sem devrait avoir pour résultat la sortie de chacune des paraphrases de la figure 2:

In Edmonton, John gave hogs to Mary.
In Edmonton, John gave Mary hogs.
In Edmonton, hogs were given to Mary by John.
In Edmonton, hogs were given by John to Mary.
In Edmonton, Mary was given hogs by John.
John gave hogs to Mary in Edmonton.
John gave Mary hogs in Edmonton.
Hogs were given to Mary by John in Edmonton.
Hogs were given by John to Mary in Edmonton.
Mary was given hogs by John in Edmonton.

Fig.2

2.2 Quelques hypothèses de base.

Dans CRITTER, la relation *anasynt* peut être vue comme une interface entre la grammaire logique réversible et la composante de transfert. Si s est l'élément non terminal correspondant à une phrase principale - et en supposant pour l'instant une traduction de type DCG des éléments non terminaux en clauses - on peut la définir de la manière suivante :

```
anasynt(String,SurfSyn,Sem) :-
                        s(S,String,[]),
                        syn(S,SurfSyn),
                        sem(S,Sem).
```

Ici, *syn* (resp. *sem*) sont des prédicats d'accès qui attribuent à l'*objet linguistique* S une structure représentant sa syntaxe de surface (resp. sa représentation sémantique).

Plus généralement, les éléments non terminaux de la grammaire ont le format uniforme suivant :

```
nt(NT) --> etc...
```

où NT est un *objet linguistique* qui contient de l'information de différents types accessible au moyen de "sélecteurs" tels que *sem*, *syn*, *cat*, *number*, etc...

L'une des propriétés importantes de cette grammaire est sa *compositionnalité* syntaxique et sémantique: tous les NT possèdent une *syn* et une *sem* et celles-ci sont construites à partir des *syn* et des *sem* des filles de NT dans l'arbre de dérivation.

Ce double aspect syntaxique/sémantique des objets linguistiques est d'une grande importance pour la réversibilité. Il devient alors possible par exemple de

diviser la tâche de synthèse dans les sous-tâches a) et b) :

a) à partir de la sémantique de NT, trouver les sémantiques potentielles de ses filles.

b) à partir de la sémantique des filles, trouver leurs syntaxes potentielles.

Avec une grammaire qui n'aurait pas cette propriété, par exemple une grammaire qui exécuterait une application directe en une seule fois de la syntaxe globale d'une phrase en une représentation sémantique, le problème computationnel de la synthèse serait bien plus difficile.

2.3 DCGs et Réversibilité.

2.3.1 Réversibilité par ordonnancement des buts.

Prenons l'exemple suivant d'une règle DCG et de sa traduction standard dans une clause définie :

Fig.3

Dans cette règle, le prédicat *combine1* condense la relation entre les trois objets linguistiques NP, VP et S. Nous ferons par la suite une description plus détaillée du fonctionnement de cette règle (section 3), mais il nous suffira pour l'instant d'indiquer que le prédicat combine1 :

- construit la syntaxe de S à partir des syntaxes de NP et VP,

- construit la sémantique de S à partir des sémantiques de NP et VP de telle façon que i) la sémantique de S et celle de VP sont immédiatement reliées (en fait, elles sont unifiables) et ii) la sémantique de NP est déterminée aussitôt que la sémantique et la syntaxe de VP sont connues.

- assure une vérification d'accord entre NP et VP.

Telle que donnée en figure 3, la règle s convient bien aux besoins d'analyse :

En *analyse*, lorsque *s* est appelé, L1 est connu (c'est la chaîne de caractères représentant la phrase à analyser), S et L3 étant non instanciés. Ainsi, il y a appel de *np* avec L1 connu et retour avec L2 et NP connus. Par conséquent, il y appel de *vp* avec L2 connu et retour avec L3 et VP connus. Ensuite, il y a retour de *combine1* avec S connu et finalement retour de s avec L3 et S connus[1].

Au contraire, le fait d'utiliser (b) pour effectuer la *synthèse* mène à un comportement computationnel catastrophique : en synthèse, lorsque *s* est appelé, seule la partie sémantique Sem_S de S est connue, L1 et L3 étant non instanciés. Ensuite, *np* est appelé avec NP, L1 et L2 complètement inconnus. La

---

[1] Nous ne faisons en fait qu'esquisser ici une preuve par recurrence informelle du fait que l'analyse "fonctionnera" aussitôt qu'un but non terminal est appelé avec une liste d'entree instanciee et qu'un appel de "combinaison" vient après des sous-buts non terminaux.

meilleure chose que l'on puisse alors espérer, sans parler de bouclage, est que *np* énumère tous les groupes nominaux avec toute leur syntaxe et toute leur sémantique, et attende que *vp* et *combine1* soient finalement appelés pour effectuer la vérification de la cohérence des sémantiques de VP et de S.

En fait, ce que nous avons besoin de faire dans le cas de la synthèse est simplement de faire en sorte que l'ordre d'appel de *np*, *vp* et *combine1* soit le suivant:

i) l'appel à *combine1(NP,VP,S)* doit venir en premier. *Au moment de l'appel, Sem_S est connu.* Au retour, Sem_VP sera connu (il est unifié à Sem S), ainsi que le fait que le sujet (suj) de VP est NP.

ii) l'appel à *vp(VP,L2,L3)* devrait venir en second. *Au moment de l'appel, Sem_VP est connu.* Au retour, VP sera entièrement instancié; sa syntaxe sera connue, et, de plus, la sémantique de chacun des groupes sous-catégorisés par la tête lexicale de VP (les arguments syntaxiques du verbe) sera connue (voir plus de détails à la section 3). En particulier, la sémantique du sujet de VP sera connue, ce qui veut dire (d'après i.) que Sem_NP sera connu. En outre, L2 sera connu: il a pour préfixe la chaîne générée par le VP.

iii) l'appel à *np(NP,L1,L2)* devrait venir en dernier. *Au moment de l'appel, Sem_NP est connu.* Au retour, NP sera entièrement instancié et L1 sera connu : il a pour préfixe la chaîne générée par le NP.

Le fait que l'efficacité, ou même la terminaison, d'un programme Prolog (du moins lorsqu'on utilise l'interpréteur standard fonctionnant en mode de recherche profondeur

d'abord) dépend fortement de l'ordre d'appel des buts et de leur état d'instanciation ne sera une révélation pour personne; la figure 4 rappelle quelques-unes des horreurs auxquelles on peut s'attendre.

```
correct(L) :-
   ok(L),
   peano(L).

peano([]).
peano(L) :-
   L = [a|L'],
   peano(L').

ok([foo]).
ok([a]).
```

A

```
correct(L) :-
   ok(L),
   peano(L).

peano([]).
peano(L) :-
   peano(L'),
   L = [a|L'].

ok([foo]).
ok([a]).
```

B

```
correct(L) :-
   peano(L),
   ok(L).

peano([]).
peano(L) :-
   L = [a|L'],
   peano(L').

ok([foo]).
ok([a]).
```

C

Comportement de A, B, C à l'invocation de : *correct(L) ?*, où L est une variable:

- en A , la solution L = [a] est trouvée, et il y a echec fini au retour-arrière, comme désiré.

- en B, il y a une boucle et aucune solution n'est trouvée.

- en C, la solution L = [a] est trouvée, mais il se produit une boucle au retour-arrière.

Fig. 4

La règle *s* ci-dessus n'est qu'une règle parmi d'autres. Ce qui est général dans la discussion précédente est le fait qu'il est nécessaire de maintenir un certain *invariant* lors du processus de synthèse:

> Lorsqu'un but non terminal *nt(NT,L1,L2)* est appelé pendant la *synthèse*, la partie sémantique NT_Sem de NT est connue au moment de l'appel. Il y a retour en temps fini de *nt* et, à ce moment-là, NT est

entièrement instancié et (le préfixe de la chaine générée par *nt* dans) L1 est connu.

En fait, la discussion antérieure de l'analyse montre qu'il y a un invariant correspondant qui est maintenu pendant le processus d'*analyse*:

> Lorsqu'un but non terminal *nt(NT,L1,L2)* est appelé pendant l'*analyse*, la liste d'entrée L1 est connue au moment de l'appel. Il y a retour en temps fini de *nt* et, à ce moment-là, NT est entièrement instancié et la liste de sortie L2 est connue.

Une analyse plus approfondie des grammaires déjà écrites pour l'analyse du français et de l'anglais nous a convaincus que l'ordonnancement des buts était bien le principal obstacle à la réversibilité et que, si nous trouvions un moyen de garantir que *nt(NT,L1,L2)* n'était appelé que sous la condition que L1 soit connu (analyse) ou Sem_NT soit connu (synthèse), nous obtiendrions alors une grammaire réversible réaliste.

Nous avons mis à l'essai deux méthodes d'ordonnancement (présentées ci-après aux sections 2.2.2 et 2.2.3). Pour des raisons expliquées par la suite, c'est la deuxième solution qui a été retenue.

2.3.2 Classement dynamique des buts par contrôle basé sur l'état d'instanciation.

La première solution a consisté à garder intact le formalisme DCG et à compiler la grammaire en un programme Prolog de la manière habituelle. Ce programme Prolog, toutefois, utilise massivement le *gelage de buts*, une notion introduite dans PrologII qui permet le lancement

asynchrone d'un but ("déclenchement de démons") lorsqu'une certaine précondition est remplie.[1]

La figure 5 donne un exemple de la façon dont les choses fonctionnent:

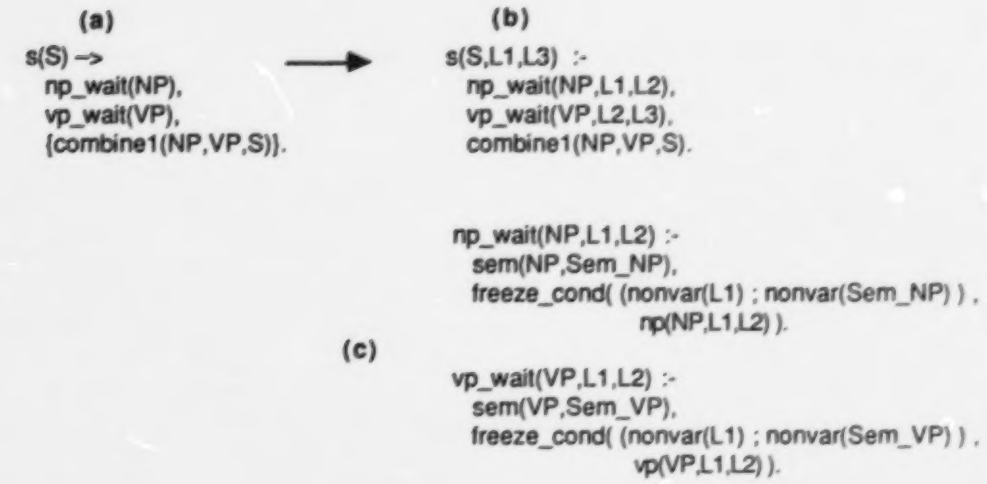

Fig. 5

Dans la grammaire, chaque élément non terminal apparaît sous deux formes: *nt* et *nt_wait*. *nt* est utilisé comme tête de règles et *nt_wait* est utilisé dans le corps de règles comme au (a) de la figure 5.

---

[1] Pour une introduction au "gelage de buts", le lecteur pourra se reporter à (Colmerauer 1982). Malheureusement, la plupart des produits Prolog sur le marché n'offrent pas encore cette fonction et nous avons dû l'implanter dans le dialecte de Prolog utilisé (Quintus Prolog) aussi efficacement que possible, c'est-à-dire *pas* par méta-interprétation!

Une règle est traduite par le compilateur DCG standard comme au (b) de la figure 5.

Finalement, les différentes instances (ie pour les différents non-terminaux *nt*) du "schéma de clauses" suivant:

```
nt_wait(NT,L1,L2) :-
    sem(NT,Sem_NT),
    freeze_cond(   (nonvar(L1) ; nonvar(Sem_NT) ),
                   nt(NT,L1,L2)
               ).
```

sont ajoutées au programme Prolog, comme au (c) de la figure 5.

La signification de ces clauses est la suivante:

> lorsque *nt_wait* est appelé, attendre avant d'appeler *nt* lui-même (ce qui implique un choix non déterministe) jusqu'à ce qu'une certaine précondition soit remplie, à savoir la condition que soit L1 soit connu ou bien que Sem_NT soit connu (où Sem_NT est la partie sémantique de NT). Aussitôt que cette précondition est remplie, lancer *nt*.

Les conséquences d'une telle formulation sont les suivantes:

> - en *analyse*, lorsque *s* est appelé - avec L1 connu - , *np* sera prêt à se déclencher immédiatement puisque sa précondition est remplie, mais *vp* devra attendre que L2 soit instancié avant de se déclencher. Pour ce qui est de *combine1*, il sera appelé immédiatement[1].

---

[1] Le terme "immédiatement" doit être compris ici de façon particulière: l'ordre exact des événements de déclenchement dépendra des détails de

- en *synthèse*, lorsque *s* est appelé - avec Sem_S connu -, ni *np* ni *vp* ne seront prêts à se déclencher, car ni Sem_NP ni Sem_VP ne sont connus à ce moment-là. Cependant, *combine1* sera appelé immédiatement et il y aura retour avec Sem_VP connu. Ensuite, *vp* se déclenchera et lorsqu'il retournera finalement (ce qui signifie que tous les sous-buts de vp auront été déclenchés), VP sera entièrement connu et L2 partiellement connu (la "chaîne du VP" sera complètement connue). En conséquence, Sem_NP sera connu et *np* pourra se déclencher. A son retour, NP sera entièrement connu et L1 partiellement connu (la "chaîne du NP" sera entièrement connue). Ainsi, la chaîne correspondant à *s* aura été générée.

Les expériences que nous avons faites dans ce cadre montrent qu'il s'agit d'une idée réalisable. En fait, nous avons pu analyser et synthétiser des phrases simples. Nous avons toutefois rencontré certaines difficultés :

- la mise au point est difficile dans le cas de l'évaluation asynchrone car il est souvent problématique d'établir le lien entre un but et son appelant. On pourrait remédier à ce problème s'il existait des fonctions de trace adaptées au gelage de buts.

- les coûts de mise en place du gelage de buts dans un langage qui n'est pas particulièrement conçu pour cette fonction ne sont pas négligeables. En outre,

---

réalisation du mécanisme de gelage de buts. Ce qui est important dans un tel schéma asynchrone n'est pas l'ordre exact des appels, mais plutôt le fait qu'un but ne sera déclenché que *si* sa précondition est remplie.

le gelage de buts étendu rend pratiquement impossible une compilation optimisée du programme Prolog obtenu à partir d'une grammaire.

- de façon plus fondamentale, il semble que le *retour arrière chronologique* (le schéma de retour arrière standard dans Prolog) ne fasse pas bon ménage avec l'évaluation asynchrone de buts: le retour arrière sur un but donné peut défaire beaucoup de travail utile qui a été fait sur des buts *appelés antérieurement* mais *logiquement non reliés*.

2.3.3 Ordonnancement statique des buts par double compilation.

• Annotations d'ordre.

Les problèmes mentionnés dans la section précédente nous ont incités à rechercher d'autres moyens d'atteindre l'ordre d'appel de buts approprié pour l'analyse et pour la synthèse.

Une solution naturelle consiste à étendre le formalisme DCG de telle manière que la notation des règles de la grammaire sépare le *contenu déclaratif* d'une règle de l'*information de contrôle* correspondant à l'ordre dans lequel les buts doivent être exécutés dans différents contextes d'évaluation. De fait, si nous considérons à nouveau la figure 3 ci-dessus, nous remarquons que l'ordre dans lequel les éléments non terminaux *np* et *vp* apparaissent dans la règle (a) joue en fait *deux* rôles différents :

- une spécification de l'ordre dans lequel *np* et *vp* apparaissent dans la chaîne, ce qui se réflète dans la traduction (b) par les associations respectives des listes différentielles L1, L2 et L3 à *np* et à *vp*.

- une spécification de l'ordre dans lequel les buts sont appelés par l'interpréteur Prolog.

De façon à distinguer entre ces deux types d'ordre, des annotations sont ajoutées dans la syntaxe des règles DCG. La figure 6 donne un exemple de ces annotations:

```
s(S) ->
  np(NP),                    #(3),
  vp(VP),                    #(2),
  {combine1(NP,VP,S)},       #(1).
```

Fig. 6

Ces annotations apparaissent à droite de chaque but et se présentent sous la forme #(n), où n est un entier. L'intention de ces annotations est de spécifier l'ordre relatif dans lequel les buts doivent être exécutés pendant la *synthèse*. Quant à l'ordre d'exécution des buts durant *l'analyse*, il est le même que l'ordre textuel des buts[1].

A partir de ces règles annotées, le compilateur de grammaire exécute deux tâches différentes, à savoir :

- traduire la grammaire en un programme Prolog d'*analyse*.

- traduire la grammaire en un programme Prolog de *synthèse*.

---

[1] Ainsi, on a un contrôle total de l'ordre d'exécution des buts en analyse et en synthèse, à ceci près que *en analyse*, l'ordre d'exécution des buts non terminaux doit être le même que l'ordre des éléments non *terminaux* dans la chaîne, et c'est d'ailleurs ce que l'on veut dans ce cas.

La figure 7 illustre cette "double" compilation :

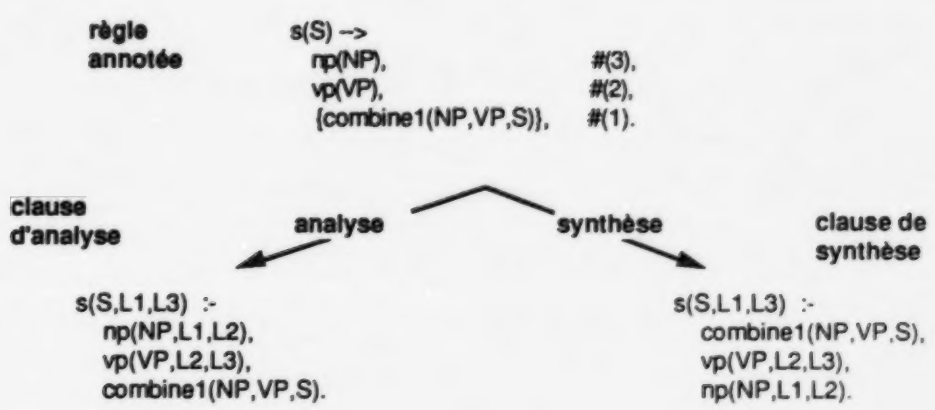

Fig. 7

Les clauses cibles d'analyse et de synthèse ainsi produites partagent le même contenu déclaratif, mais les appels de buts se font dans l'ordre approprié.

- **Traitement de la récursivité à gauche.**

Il y a d'autres insuffisances du formalisme DCG du point de vue de la réversibilité qu'il est utile de corriger.

L'une de ces lacunes a trait à des règles du type :

```
np(NP) -->
 np(NP'),
 pp(PP),
 {combine2(PP,NP',NP)}.

np(NP) -->
 n1(N1).
```

Fig. 8

La règle[1] *np2* ci-dessus vise à intégrer dans le *np* un nombre non spécifié de modificateurs prepositionnels (*the hotel in Vancouver with a Chinese roof*). *n1* est le *np* "nu" (the hotel) et *combine2* construit les structures syntaxique et sémantique du NP à partir des structures du PP et du NP'.

Le problème évident avec cette règle, cependant, est qu'elle est "récursive à gauche": bien que sa lecture *déclarative* soit parfaitement naturelle et représente directement la signification visée de la règle, la traduction DCG standard conduira à une boucle dans le cas de l'analyse.

Pour cette raison, aucun programmeur DCG n'écrira ces règles de la manière illustrée à la figure 8, mais il les transformera plutôt en des règles équivalentes qui, bien que plus complexes, produiront le comportement computationnel voulu (voir la figure 9 ci-après).

Toutefois, il est intéressant de noter que la règle de la figure 8, lorsqu'elle est utilisée en synthèse, fonctionnera parfaitement bien si seulement on veille - comme dans l'exemple précédent (vp) - à ordonner le but *combine2* avant les buts *pp* et *np*[2].

Heureusement, il y a moyen de tourner cette difficulté : le grammairien doit avoir la possibilité d'écrire la règle *np*

---

[1] Cette règle sert seulement d'exemple et ne prétend pas à l'adequation linguistique.

[2] Il est assez ironique que - d'après notre expérience - si on écrit la règle np de manière à éliminer la récursivité à gauche dans l'analyse et que l'on essaye de réordonner les buts pour les besoins de synthèse, d'autres difficultés réapparaissent qui rendent tout le processus quelque peu absurde!

de façon naturelle et le compilateur aura la responsabilité d'exécuter les transformations nécessaires pour l'analyse et pour la synthèse.

C'est précisément ce qui est fait dans le système CRITTER, comme l'illustre la figure 9:

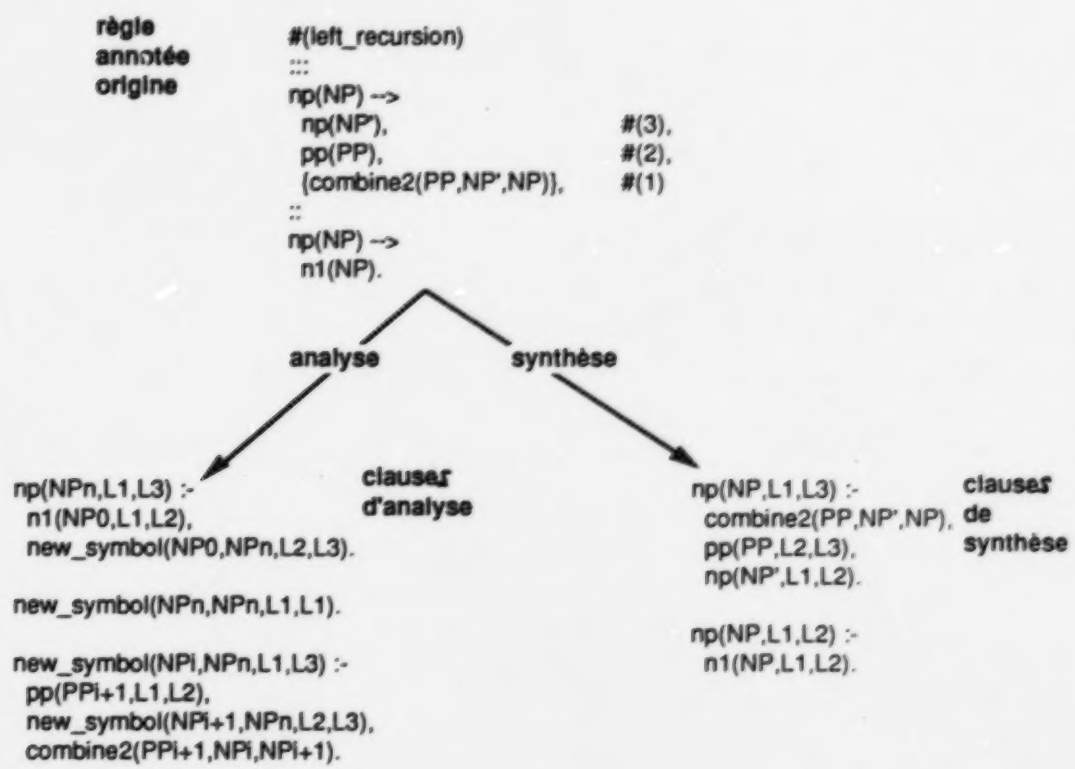

Fig. 9

Dans la réalisation actuelle, il faut indiquer au compilateur où se trouve la règle récursive à gauche et signaler le bloc approprié de règles. Ceci explique la présence d'une directive *#(left_recursion)*, ainsi que celle des "séparateurs" *:::* et *::* . Le traducteur d'analyse introduira alors un nouveau symbole non terminal "fictif" *new_symbol* et exécutera une transformation produisant les clauses cibles d'analyse montrées dans la figure. L'espace manque ici pour détailler la transformation effectuée, mais

il est possible de démontrer formellement que les clauses résultantes gardent la signification déclarative de la règle inchangée[1].

Quant aux clauses de synthèse, elles sont semblables aux règles originales, en dehors du fait que les annotations d'ordre dans ces règles ont servi à ordonner les sous-buts, comme dans le cas de la figure 7.

- **Autres dispositifs notationnels.**

Tirant partie de la souplesse obtenue grâce à l'introduction d'un compilateur sur mesure, nous avons ajouté une notation semblable à la notation fonctionnelle utilisée dans les grammaires d'unification (Sag et al., 1986), c'est-à-dire la possibilité de renvoyer aux sous-structures d'un objet linguistique au moyen d'une notation "pointée".

Par exemple, la règle de la figure 6 s'écrit en réalité de la manière suivante :

```
s(S) -->
     np(NP),                          #(3),
     vp(VP),                          #(2),
     {S.head = VP.head,
      S.head.subcat.suj = NP,
      S.sem_form = VP.sem_form,
      NP.number = VP.number,
      . . .
     },                               #(1).
```

---

[1] Le processus est semblable aux techniques qui permettent d'amener une grammaire hors-contexte sous une certaine forme normale. La partie délicate ici, toutefois, est la manière dont les contraintes entre les *variables* doivent être transformées.

Le compilateur (dans le cas de l'analyse aussi bien que de la synthèse) remplacera par exemple "S.head" par une variable Shead, ajoutera à la clause générée la contrainte "head(S,Shead)" (et fera de même pour les autres cas) et effectuera les unifications exigées par les égalités figurant dans la règle.

3. Un Exemple.

Nous allons donner maintenant un exemple qui illustre, quoique de manière simplifiée, quelques traits essentiels de notre façon d'aborder la réversibilité. Nous allons nous concentrer sur le problème de l'analyse et de la synthèse de phrases telles que (1) :

(1) Jack seems to like Jill.

Le verbe "seem" appartient à une classe de verbes dits "de montée". Selon une analyse bien établie de ces structures, la représentation sémantique associée à (1) sera quelque chose comme (2) :

(2) seem'(like'(jack, jill))

C'est-à-dire que le sujet de surface "Jack" n'est pas interprété comme un argument de seem'. Seem' est plutôt vu comme un opérateur propositionnel et "Jack" est un argument dans la prédication exprimée par le complément infinitival.

Dans notre modèle grammatical, ces particularités de l'application syntaxique/sémantique sont décrites dans la composante lexicale, de façon assez similaire à celle de (Pollard et Sag, 1988). Le lexique attribue à "seem" une structure de traits VB ayant les propriétés suivantes :

(3)

VB.cat = vb	(a)
VB.cit_form = seem	(b)
VB.head.subcat = [NP, VCOMP]	(c)
NP.cat = np	(d)
VCOMP.cat = vp	(e)
VCOMP.head.form = infinitive	(f)
VCOMP.head.sem_form = A	(g)
VCOMP.head.subcat = [VPSUBJ | X]	(h)
VPSUBJ.head.sem_form = NP.head.sem_form	(i)
VB.head.sem_form = seem'(A)	(j)
inflect(seem, NP.agree, VB.inflected_form)	(k)

Dans la description (3), les équations (c) à (j) attribuent à "seem" une structure de tête *head* qui, sous l'action des règles syntaxiques, va percoler dans l'arbre d'analyse jusqu'à ce que le constituant maximal dont "seem" est la tête soit atteint. Cette structure de tête contient une sous-structure "subcat" [NP, VP] (décrite plus en détail dans (d) à (i)) et une forme sémantique seem'(A). Le prédicat "inflect" établit la forme fléchie du verbe sur la base de sa forme de citation ainsi que ses attributs d'accord (pour plus de simplicité, nous ne tenons pas compte ici des temps).

La sous-structure "subcat" rend compte des dépendants syntaxiques de "seem" à la fois au niveau de leurs propriétés syntaxiques et au niveau de leur contribution à la structure sémantique englobante. Plus précisément, "seem" nécessite un groupe nominal sujet et un complément infinitival. Ce dernier est associé à un objet sémantique A qui se trouve être unifié à l'argument unique de seem' -- like'(jack,jill) dans le cas de la phrase (1). Les équations (h) et (i) lient la valeur sémantique du sujet de

"seem" à la forme sémantique sem_form associée à la case sujet du verbe principal du complément infinitival. Ceci rend compte du fait que ce complément est "contrôlé " par le sujet de "seem". Pendant la synthèse du complément, la valeur sémantique attachée à la case sujet -- 'jack' dans le cas de la phrase 1 -- sera connue. Du même coup, la sem_form du sujet de "seem" sera instanciée.

La spécification lexicale de (3) donne un exemple d'un trait important des modèles "lexicalistes": dans un groupe syntaxique, l'application entre les configurations syntaxiques (ou fonctions syntaxiques) et les positions des arguments dans les structures sémantiques est déterminé par les propriétés lexicales de la tête.

Nous écrivons des règles de grammaire telles que les suivantes :

```
(4)
s(S) -->
    {S.cat = s},
    {S.daughters = [NP, VP]},
    {S.head = VP.head},
    {S.head.subcat = [NP | _]},
    np(NP),
    vp(VP).


np(NP) -->
    det(DET),
     ... etc.

np(NP) -->
    {NP.cat= np},
    {NP.daughters = [NPR]},
    {NP.head = NPR.head},
    npr(NPR).

vp(VP) -->
    {VP.cat = vp)},
    {VP.daughters = [VB, COMP])},
    {VP.head = VB.head},
    {VP.head.subcat = [_, COMP]},
    vb(VB),
    complement(COMP).

complement(COMP) -->
    np(COMP).
complement(COMP) -->
    pp(COMP).
complement(COMP) -->
```

```
vp(COMP).
```

La règle s, par exemple, sera compilée en la clause suivante :

```
(5)  s(S, L1, L2) :-
        cat(S, s),
        daughters(S, [NP, VP]),
        head(S, H), head(VP, H),
        subcat(H, [NP | _]),
        np(NP, L1, L3),
        vp(VP, L3, L2).
```

Le programme résultant pourra *analyser* la phrase (1) de manière efficace. Un appel au but s(S, "Jack seems to like Jill", "") réussira et l'instanciation de S donnera un objet ayant les propriétés voulues. En particulier, il sera établi que :

```
(6)      S.head.sem_form = seem'(like'(jack, jill))
```

Dans l'obtention par l'analyseur de ce résultat, l'identité (i) de l'entrée lexicale (3) joue un rôle critique. Le fait de lier la sem_form du sujet de "seem" à la sem_form du sujet de son complément produit l'effet approprié de "descente" du sujet.

Supposons maintenant que nous voulions utiliser le même programme pour la *synthèse*. Dans ce cas, nous devons démontrer s(S,L1,"") et tout ce que nous savons au sujet de S est que (6) est vrai. Le problème, toutefois, c'est que la grammaire commence par déclencher np(NP,L1,L2) alors que NP, L1 et L2 sont toutes des variables non instanciées. Il est bien possible que le programme se mette à boucler lors de l'énumération de np bien formés puisque quelques-unes des règles d'expansion des np vont probablement permettre

des structures de modification récursives. En général, il est tout à fait illogique sur le plan computationnel d'essayer de faire la synthèse d'une sous-structure dont la forme sémantique n'est pas encore connue.

Il n'y a aucun moyen de déterminer la sem_form de ce groupe nominal par un simple examen de la sem_form de la phrase. Ce qu'il faut faire, c'est de trouver dans le dictionnaire un élément lexical dont la sem_form corresponde à celle de la phrase; l'entrée appropriée du dictionnaire déterminera alors la distribution des arguments sémantiques dans les positions syntaxiques. Le prédicat seem' donne accès à l'entrée du verbe "seem" où il est stipulé : 1) que l'argument unique du prédicat seem' doit être associé à un complément infinitival du verbe et 2) que le sujet du verbe est lié à la case sujet du complément infinitival.

Avec d'autres verbes, la situation peut être différente. La forme active d'un verbe transitif simple associe généralement le premier argument du prédicat correspondant à la position sujet. Avec la forme passive du même verbe, c'est au contraire le deuxième argument qui est associé à la position sujet. Avec les adjectifs dits de "tough-movement", le sujet syntaxique est associé à une case syntaxique non sujet dans le complément infinitival :

(7) a) Mary is difficult to convince.
b) difficult'(convince'(one, mary))

Étant donnée la nécessité de connaître la sem_form d'un groupe avant de le synthétiser, il n'y a pas d'autre possibilité que celle de synthétiser le groupe verbal avant le groupe nominal sujet. Pour cela, nous annotons la règle s comme suit :

```
(8) s(S) -->
       {S.cat = s,
        S.daughters = [NP, VP]),
        S.head = VP.head,
        S.head.subcat = [NP | _] ),        #(1),
       np(NP),                             #(3),
       vp(VP),                             #(2).
```

Le compilateur décrit à la section 2.3.3. compilera la règle (8) dans la clause de synthèse (9) :

```
(9)  s(S, L1, L2) :-
       cat(S, s),
       daughters(S, [NP, VP]),
       head(S, H), head(VP, H),
       subcat(H, [NP | _]),
       vp(VP, L3, L2),
```

np(NP, L1, L3).

Le but s(S,L1,L2), seule l'identité (6) étant "connue", déclenchera la séquence suivante d'unifications :

La règle s est appelée :

(a) S.head.sem_form = seem'(like'(jack,jill))
(b) S.daughters = [NP, VP]
(c) VP.head = S.head

La règle vp est alors appelée:

(e) VP.daughters = [VB, COMPL]
(f) VB.head = VP.head
(g) VP.head.subcat = [NP, VCOMP]

Vp appelle la règle vb qui effectue une consultation de dictionnaire:

(h) VB.cat = vb
(i) VB.cit_form = seem
(j) VB.head.subcat = [NP, VCOMP]

"Seem" prend un complément infinitival dont la sem_form est liée à l'argument unique de seem':

(k) VCOMP.head.sem_form = like'(jack,jill)

La case sujet du complément infinitival de "seem" et le sujet de "seem" partagent le même contenu sémantique:

(l) VCOMP.head.subcat = [NP2 | X]
(m) NP2.sem_form = NP.sem_form

Comme la forme syntaxique du sujet de "seem" n'est pas encore connue, les valeurs d'accord du verbe ne sont pas connues. Le processus qui calcule la flexion le verbe doit être différé[1]:

(n) VB.inflected_form = ??

WAIT jusqu'à ce que NP.agree soit instancié:
inflect(seem,NP.agree,VB.inflected_form)

Au retour de la règle vb, vp appelle la règle de complément qui fait un nouvel appel à la règle vp afin de synthétiser le complément infinitival de "seem":

(o) VCOMP.head = VB2.head
(p) VB2.form_cit = like
(q) VB2.head.subcat = [NP2, NP3]
(r) NP2.sem_form = NP.sem_form = jack
(s) NP3.sem_form = jill

La synthèse du vp "to like Jill" est terminée. Maintenant, la sem_form du sujet de "seem" est connue : elle a été identifiée à la sem_form de la case sujet de "like" qui a elle-même été liée au premier argument du prédicat like', à savoir 'jack'.

Nous revenons maintenant au vp le plus englobant. Sa synthèse est alors complète, à l'exception du processus de flexion de "seem" qui reste sur la liste d'attente des buts "gelés".

---

[1] Cela se fait à l'aide du mécanisme de "gelage de buts" (voir la section 2.3.2.)

De retour au niveau de s, le but associé au np sujet se déclenche:

(t) NP.cit_form = 'Jack'
(u) NP.agree.pers = p3
(v) NP.agree.numb = sing

La synthèse du np sujet est terminée. Comme NP.agree est maintenant connu, le but différé peut être "dégelé" (c'est-à-dire appelé):

(n') *MELT:*
inflect(seem,NP.agree,VB.inflected_form)
VB.inflected_form = seems

(o) le but s(S,L1,L2) réussit avec:
L1 = "Jack seems to like Jill".

4. Conclusion.

Après avoir expérimenté différentes approches pour résoudre le problème de la réversibilité dans le cadre des grammaires logiques, la solution la plus pratique s'est avérée être la suivante: les règles sont annotées de directives qui imposent un ordonnancement des buts ainsi que des transformations globales au moment de la compilation ("at compile-time"), et qui réaultent en un programme d'analyse et un programme de synthèse partageant le même contenu grammatical.

Cette approche donne au linguiste le contrôle total du comportement computationnel de la grammaire - à la fois en

analyse et en synthèse - sans compromettre la clarté déclarative des spécifications grammaticales.

Grâce à l'implantation de cette idée, le système CRITTER peut actuellement traduire des phrases modérément complexes de l'anglais au français et vice versa, à l'aide d'une seule grammaire pour chaque langue (la réversibilité de tout le système CRITTER dépend également de la réversibilité d'autres composants que les grammaires, tels les composants morphologiques et de transfert qui n'ont pas été décrits dans ce document).

L'utilisation bidirectionnelle du système - à la fois en mode de traduction et particulièrement en mode unilingue (analyse et "resynthèse" d'une phrase pour produire ses paraphrases) - s'est avérée avoir certaines conséquences imprévues mais intéressantes:

- Il arrive souvent que des erreurs ou des omissions dans la description grammaticale, qui passent facilement inaperçues en mode d'analyse (puisqu'on n'est pas naturellement confronté à des phrases qui les exposent), deviennent immédiatement apparentes lorsque des paraphrases grammaticalement incorrectes sont synthétisées.

  Une grammaire réversible agit donc comme un puissant outil de mise au point pour elle-même!

- Une grammaire réversible (RG) doit nécessairement être plus proche de l'adéquation observationnelle que ne le seraient deux grammaires différentes, l'une d'analyse (AG) et l'autre de synthèse (SG). En effet une grammaire d'analyse (AG) peut se permettre d'être sur-permissive dans les chaînes qu'elle accepte, puisqu'elle peut faire confiance dans une large mesure à la grammaticalité du texte d'entrée[1]. Quant à une grammaire de synthèse (SG), elle peut se permettre d'être sous-permissive dans les chaînes qu'elle génère car elle doit uniquement garantir la grammaticalité de la sortie, mais non que toutes les variations possibles dans l'expression du même contenu seont produites.
Par contre, le fait de travailler avec une grammaire réversible (RG) exerce une pression certaine sur le linguiste qui est obligé d'affiner ses descriptions pour qu'elles correspondent à la réalité linguistique. Il est difficile à ce stade d'évaluer si ce facteur sera vu en dernier ressort comme un handicap ou comme un bonus.

Comme les remarques précédentes l'indiquent, le problème de la réversibilité présente en réalité deux facettes différentes :

Le problème de la réversibilité *computationnelle*: comment exécuter computationnellement des tâches d'analyse et de synthèse à partir de la même grammaire déclarative? Ce problème est en majeure partie résolu dans le cas des grammaires DCG par les

---

[1] Bien sûr, cela ne va pas sams problèmes: la sous-spécification de AG finira par se trahir par la production de fausses ambiguïtes syntaxiques, même lorsqu'on peut se fier à la bonne formation du texte analysé.

techniques présentées ici. Toutefois, il reste encore certaines recherches à faire pour étendre ce traitement aux dépendances non bornées, telles qu'elles sont traitées par exemple dans les grammaires d'extraposition (Pereira 1981).

Le problème de la réversibilité *linguistique* : est-il théoriquement correct d'utiliser une seule et même grammaire pour décrire les aspects linguistiques de l'analyse et de la synthèse et, dans ce cas, est-ce pratiquement faisable ou bien cela impose-t-il des contraintes peu réalistes à l'activité de description linguistique? Sur ce point, que la présente communication a à peine effleuré, il reste encore beaucoup de travail à faire.

Remerciements.

Nous remercions Elliott Macklovitch, Jean-Luc Cochard, Ron Ferguson, François Perreault et Michel Simard de leurs commentaires et suggestions.

Références.

Appelt, D. (1987) "Bidirectional Grammars and the Design of Natural Language Generation Systems", in Theoretical Issues in Natural Language Processing, New Mexico State University.

Colmerauer A. (1982) PROLOG II : Manuel de référence et modèle théorique. Groupe d'Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille.

Isabelle P., Dymetman M., Macklovitch E. (1988) "CRITTER: a Translation System for Agricultural Market Reports", to appear in Proceedings of COLING-88.

Landsbergen, J (1987) Montague Grammar and Machine Translation. Philips Research M.S. 14.026, Eindhoven.

Pereira, F. (1981) "Extraposition Grammars", in Computational Linguistics 7.4, 243-256.

Pereira F., Warren D.H.D. (1980) "Definite Clause Grammars for Natural Language Analysis", Artificial Intelligence, 13, 231-278.

Pollard, C., Sag, I. (1988) Information-Based Syntax and Semantics, vol. 1: Fundamentals, CSLI lecture Notes no 13, CSLI, Stanford University.

Sag I., Kaplan R., Karttunen L., Kay M., Pollard C., Shieber S., Zaenen A. (1986) "Unification and Grammatical Theory", in Proceedings of the West Coast Conference on Formal Linguistics, Stanford Linguistics Association, Stanford University.

Zaharin Y, (1987) "String-Tree Correspondence Grammar: a declarative formalism for defining the correspondence between strings of terms and tree structures", in Proceedings of the Third Conference of the European Chapter of the Association for Computational Linguistics, Copenhagen.